Süsteemprogrammeerimine keeles C

Dünaamiline mäluhaldus

Heap

- Dünaamilise mälu jaoks mõeldud mälu osa
- Piiratud kernelis pointeriga brk
- Mälu hõivamine ja vabastamine:
  void *sbrk()
- Otse seda ei kasutata
  alloc(), malloc(), calloc(), free()
- Mälu haldusest veidi hiljem

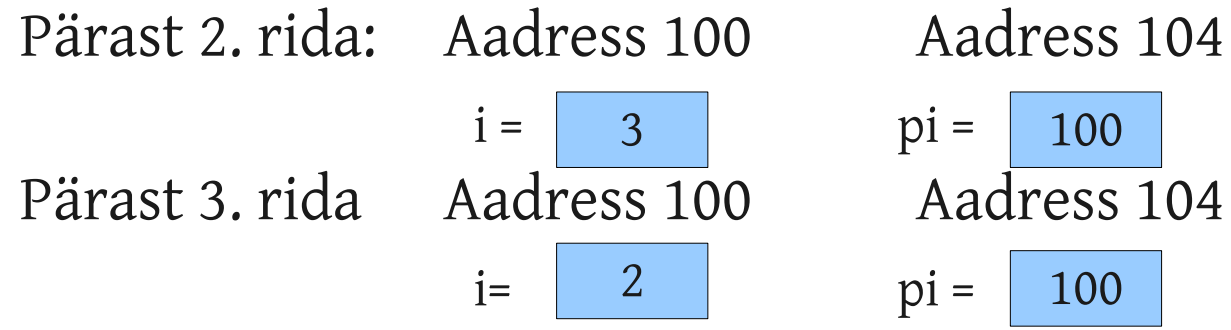
Aadressid ja pointerid (1)

- Kõik arvutis olevad objektid paiknevad mingil aadressil
- Mõndadele C programmi objektidele saab nende mälus paiknemise aadressi järgi viidata
  - *Expression*&var saab väärtuseks var-i aadressi
- Ajutist mäluaadressi omavad objektid (numbrikonstandid, liittehted) pole & operaatoriga kasutatavad.
- Aadresse saab hoida pointerites

Aadressid ja pointerid (2)

- Kui pvar on pointer, milles on aadress, võtab * operaator (*dereference, indirection*) sellel aadressil paikneva sisu (tehe: *pvar).
- * operaatorit kasutame ka pointerite defineerimisel

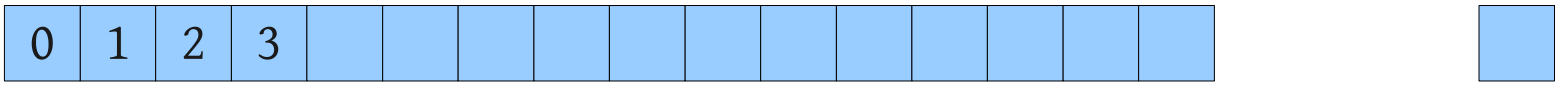
```
1:  int i, *pi;          // *pi - pointer integerile

2:  i = 3; pi = &i;      // now (*pi == 3)
3:  *pi = 2;             // now (i == 2)
```



Aadressid ja pointerid (3)

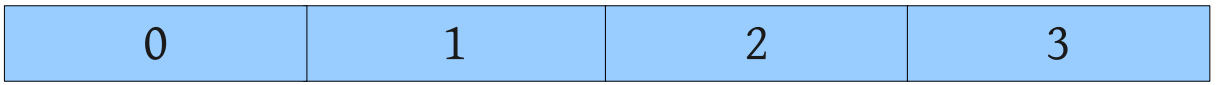
- Pointeris olevale aadressile * operaatori rakendamisel peame teadma andmete tüüpi
  - Seda seepärast, et erinevad andmetüübid võtavad erineva hulga mälu (char, int, float, double).

```
char c[N]; char *pc = &c[0]; // *(pc+1) == c[1];
```

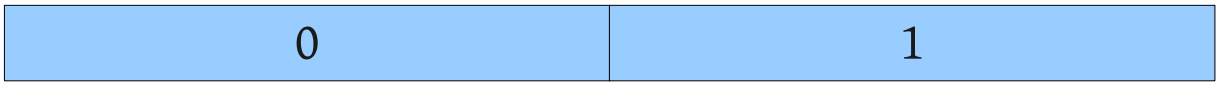


- 1 bait

```
int i[N]; int *pi = &i[0]; // *(pi+1) == i[1];
```




```
double d[N]; char *pd = &d[0];// *(pd+1) == d[1];
```



Pointerite võrdlemine

- Pointerite võrdlemine ei ole reeglina mõttekas tegevus
- Erand, mis reeglit kinnitab:
  - Sama massiivi elemendid paiknevad mälus järjest ja nende võrdlemine võib olla mõttekas
- Teine erand, mis reeglit kinnitab:
  - Võid alati kontrollida kas mingi pointer on NULL

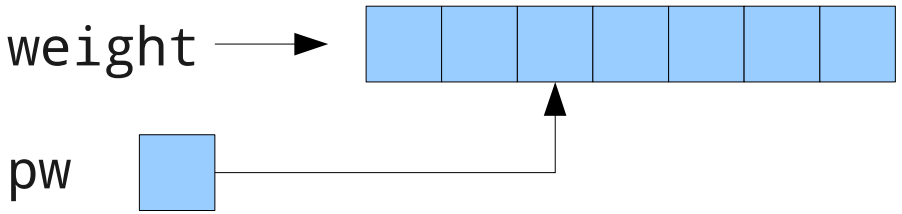
Pointer ja massiiv

- Kui on olemas definitsioon: **`double weight[LEN], *pw;`** kehtib järgnev.
- **`weight[i]`** on *expression*, mis viitab väärtusele, mis on salvestatud massiivi i-nda elemendina
- **`weight`** on pointer tüüpi *expression*, mis viitab massivi esimesele elemendile
  - Järeldus: (weight == &weight[0]) on **alati** tõene.
- C kompilaator teeb **`weight[i]`** korral sisemise teisenduse, mille tulemuseks on **`*(weight + i)`**.
- Pärast omistust **`pw = weight`** on **`pw[2]`** sama väärtusega, mis **`weight[2]`**
- **`pw`** ja **`weight`** põhierinevus on see, et **`weight`** on konstant ja seda ei saa muuta, samas kui **`pw`**-d võib muuta

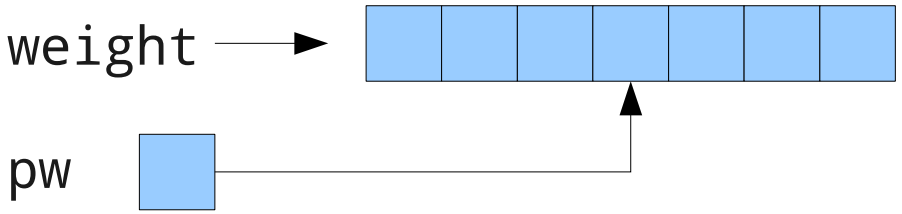
Pointer ja massiiv (Näited)

- Näide:

```
pw = weight + 2
```




```
pw++; // pw[0] on weight[3]
```



- Samas:

```
weight++;  /* VIGA! */
```

Pointerid ja funktsiooni argumendid

- C keeles saab argumente ainult koopiana anda (*pass-by-value*)
- See tähendab, et funktsioonile antud muutuja funktsiooni töö ajal ei muutu.
- Pointeritega saab sellest mööda hiilida:

```
Example:
void Swap(int *a, int *b)
{
  int t = *a; *a = *b; *b = t;
}
int i = 3, j = 4;
Swap(&i, &j);
```

- ülaltoodu on väljakutse viitamisega (*call by reference*)

Massiiv funktsiooniargumendina

- Massiiv on erand koopia reeglist.
- Kui funktsiooni argumendiks on massiiv, ei kopeerita kogu massiivi, vaid edastatakse aadress.
  - St sisuliselt ei ole tegemist erandi, vaid faktiga, et kopeeritakse massiivi pointer
  - int func vec[SIZE] puhul on func(vec) ja func(&vec[0]) samaväärsed;
  - definitsioonis on func(int *arr) ja func(int arr[]) identsed
  - massiivi osa edastamisel on samaväärsed näiteks func(vec+2) ja func(&vec[2])

Topeltviitamise meenutus

- Võtame

```
int a;
int *pa;
int **ppa;
```

- Mis on &a tüüp?
  - Pointer täisarvule (int *pa)
- Mis on &pa tüüp?
  - Pointer täisarvu pointerile (int *pa)
- Kas pärast omistust pa = &a on korrektsed?

```
*ppa = pa;
*ppa = &a;
int **ppa = &&a;
```

Dünaamiline mäluhaldus(1)

- C lubab programmil *heap*ist jooksvalt mälu võtta. Seda piirab vaid jooksmise ajal saadaval oleva mälu hulk.
- Selleks on 3 (stdlib.h all defineeritud) funktsiooni:

```
void *malloc(mitu_baiti_anda);
void *calloc(mitu_on, mis_suurusega);
void *realloc(pointer, mitu_baiti_anda);
```

- Kui mälu hõivamine ei õnnestu, tagastub NULL
- Kuna tagastatakse void tüüpi pointer, tehakse *cast*.

```
int *pi = (int*)malloc(5*sizeof(int)); /* või */
int *pi = (int*)calloc(5, sizeof(int));
pi = (int*)realloc(pi, 10*sizeof(int));
```

Dünaamiline mäluhaldus(2)

- Millal seda tarvis läheb?
  - Näiteks, kui massiivi suurus antakse programmile argumendina
  - Üldreegel: Alati, kui programm kompileerimise ajal ei tea kui palju mingi asi ruumi võtab
- OLULINE: Pärast võetud mälu kasutamist tuleb see süsteemile tagasi anda:

```
void free(void*);
```

- Kui seda ei tehta, tekib mäluleke: kasutamata mälu tundub süsteemile "lukus" ja seda ei saa kasutada.

Mäluhalduse näide

- Korrektne näide

```
int *vec;
if ((vec=(int*)malloc(ARR_LNG*sizeof(int)))==NULL) {
   fprintf(stderr, "cannot allocate\n");
   exit(1);
}
if ((vec = (int*)realloc(vec, NEW_ARR_LNG*sizeof(int)))
   == NULL) {
   fprintf(stderr, "cannot allocate\n");
   exit(1);
}
```

- Üks võimalik viga: ripakil pointer (*dangling pointer*), tekib siis, kui pointer viitab juba vabastatud kohta.

Mäluhalduse halb näide

- Vea näide:

```
int *vec, *new_vec;
if ((vec=(int*)malloc(ARR_LNG*sizeof(int)))==NULL) {
   fprintf(stderr, "cannot allocate\n");
   exit(1);
}
if ((new_vec =
     (int*)realloc(vec, NEW_ARR_LNG*sizeof(int)))
       == NULL) {
   fprintf(stderr, "cannot allocate\n");
   exit(1);
}
```

- vec näitab nüüd kuhu juhtub

Veel üks ripakil pointer

```
char *foo(char *s) {
   char buf[100];
   strncpy(buf, s, 99);
   return buf;
}

main() {
   char *t;
   t=foo(“Hello”); // t on ripakil
}
```

Mälu haldamise põhimõtteid

- Malloc jagab mälu omavahel lingitud blokkideks
- Vabad blokid
- Segregeerimine klassideks
- Segmenteerumisprobleem
- (First-fit, best-fit, round-robin)

Pointerid funktsioonidele

- Võib tekkida situatsioon, kus tuleb välja kutsuda funktsioon, kuid pole teada milline

```
void *v1, *v2;
if (compare (v1, v2) == 1) { ...
```

v1 võib viidata stringile või integerile. Tuleks kutsuda välja objektitüübile vastav funktsioon...

```
enum type {INT, STR};
int (*compare)(void*, void*); /* pointer funktsioonile */
...
switch (type) {
   case INT: compare = &num_compare; break;
   case STR: compare = &strcmp; break; }
if ((*compare)(v1,v2) == 0) { ... /* või "!
   compare(v1,v2)"*/
```

Pointerid funktsioonidele

- Teine situatsioon, kus vajame pointerit funktsioonile on see, kui kasutame funktsiooni argumendina teisele funktsioonile

```
void string_manipulation(char *s, int (*chr_mnp)(int))
{
  while ( *s !='\0') {
   *s = chr_mnp(*s);
   s++;
  }
}
/* Use of that function * /
char str[10] = "aBcD";
...
string_manipulation(str, tolower);
```

Ühemõõtmelised massiivid

- Fikseeritud (*stack*is) ja dünaamilised (*heap*is) massiivid käituvad täpselt ühtmoodi, kui deklareerimine välja arvata.

```
int vec[100]; /* staatiline */
```

```
int *vec;    /* dünaamiline */
vec = (int*)malloc(sizeof(int)*100);
```

```
/* kasutamine */
vec[70] = 1; /* või */ *(vec+70) = 1 ;
```

```
/* initsialiseerimine */
for (i=0;i<100;i++) vec[i]=0; /* ei ole tõhus */

int *ptr = vec; int *end = vec+99; *end = 0;
while(ptr != end) *ptr++ = 0;
```

```
/* funktsiooni prototüüp */
void func(int * ptr);  /* või */
void func(int ptr[]);
```

Fikseeritud kahemõõtmelised massiivid

- ***Stack*is olevad massiivid**
  Hõivamine: int fixed[50][100];
- **Ligipääs:** fixed[5][9] = 1; /* või */
  fixed[0][5*100+9] = 1; /* või */
  fixed[1][4*100+9] = 1; /* jne */
- **Initsialiseerimine:**

```
for(i=0;i<50;i++) for(j=0;j<50;j++) fixed[i][j] = 0; /* ebatõhus */

int *ptr = fixed[0]; int *end = fixed[49]+99; *end = 0;
    while(ptr != end) *ptr++=0;
```

- **Funktsioonile andmine:**
  Prototüüp: void func(int fixed[50][100]);

Dünaamilised kahemõõtmelised massiivid

- Asuvad *heap*is – see on paindlikum

Hõivamine

```
int **dynamic;
dynamic = (int**)malloc(sizeof(int*)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i]=dynamic[i-1]+100;
```

Ligipääs

```
dynamic[5][9] = 1; /* või */
dynamic[0][5*100+9] = 1;      /* või */
dynamic[1][4*100+9] = 1;      /* jne... */
```

Initsialiseerimine

```
int *ptr = dynamic[0];
int *end = dynamic[49] + 99; *end = 0;
while (ptr !=end) *ptr++=0;
```

Prototüüp

```
func(int** vec);
```

Massiividest veel

- Massiivid tuleb initsialiseerida, nad ei ole vaikimisi 0-dega täidetud
- Massiivi mälu tuleb vabastada, seda tehakse vastupidiselt hõivamisele:

```
free(dynamic[0]);
free(dynamic);
```

- Dünaamiliselt hõivatud massiivi "dynamic" indekseerimisel kujul dynamic[i][j], pole vaja korrutada.
- Dünaamiline 3d massiiv on pointerite massiiv, mis viitab dünaamilistele 2d massiividele.

Meeldetuletus teemal *casting*

- Tüübiteisendused sunnivad tehete tulemusi olema mingites kindlates tüüpides (teisendamine vägisi)

```
int i = 2; double d = 1.7;
i * d;  /* tüüp double, väärtus 3.4 */
(int)(i*d) /* tüüp int, väärtus 3 */
i*(int)d /* tüüp int, väärtus 3 */
```

- Sunnime funktsiooniargumendi õigesse tüüpi:

```
d = sqrt((double)i);
```

Pointerite teisendamine

- Mõnikord tekib situatsioon, kus peame defineerima pointeri teadmata mis tüübile ta viitab
- Selliseks üldiseks pointeritüübiks on void tüüpi pointer. void tüüp teiseneb vajadusel ise.

```
int a[10]; void *pv = &a[1]; int *pi=&a[2];
int i = pi - pv; /* hoiatab, kuid saab hakkama */

int i; double d,e;
void *pv0=&i, *pv1 = &d;
e = *pv0 + *pv1;      /* ei kompileeru */
e = *((int*)pv0)+*((double*)pv1); /* töötab */
```

Täheldusi kontrolltööks

- i++, ++i
- static
- a[1], a+1, *a+1, *(a+1), &a[1]
- {}
- x ? 1 : 0;
- 2,3 2.3
- case

Viimane slaid

- Alustage nihelemisega
- Lobisege ja lahkuge